Mining Modal Scenario-Based Specifications from Execution Traces of Reactive Systems

David Lo[†] Shahar Maoz[‡] Siau-Cheng Khoo[†]

[†]{dlo,khoosc}@comp.nus.edu.sg
Dept. of Computer Science
National University of Singapore

[‡]shahar.maoz@weizmann.ac.il
Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel

ABSTRACT

Specification mining is a dynamic analysis process aimed at automatically inferring suggested specifications of a program from its execution traces. We describe a novel method, framework, and tool, for mining inter-object scenario-based specifications in the form of a UML2-compliant variant of Damm and Harel's Live Sequence Charts (LSC). LSC extends the classical partial order semantics of sequence diagrams with temporal liveness and symbolic class level lifelines, in order to generate compact and expressive specifications. The output of our algorithm is a sound and complete set of statistically significant LSCs (i.e., satisfying given thresholds of support and confidence), mined from an input execution trace. We locate statistically significant LSCs by exploring the search space of possible LSCs and checking for their statistical significance. In addition, we use an effective search space pruning strategy, specifically adapted to LSCs, which enables efficient mining of scenarios of arbitrary size. We demonstrate and evaluate the utility of our work in mining informative specifications using a case study on Jeti, a popular, full featured messaging application.

Categories and Subject Descriptors: D.2.1 [Software Engineering]:Requirements/Specifications–Tools;D.2.7 [Software Engineering]:Distribution, Maintenance and Enhancement–Restructuring, reverse engineering and reengineering
General Terms: Algorithms, Design, Experimentation
Keywords: Specification Mining, Dynamic Analysis, UML Sequence Diagrams, Live Sequence Charts

1. INTRODUCTION

Analyzing the behavior of software systems, in order to aid program comprehension, reduce their maintenance costs, and improve their quality, is a complex and challenging task. Having incorrect, incomplete, or outdated documented specifications, as a result of short time-to-market constraints, changing requirements, and poorly managed product evolution, reduces comprehension of the code base, increases maintenance costs, and adds challenges towards verification of their correctness. One approach to address this challenge is to automatically infer specifications of a system from its execution traces by a dynamic analysis process referred to as *specification mining* (see, e.g., [6, 19]).

In this work, we focus on mining specifications of reactive systems, discrete event systems which maintain ongoing interaction with their environment, and on their behavioral specification using inter-object scenarios. Scenarios, depicted using variants of sequence diagrams, are a popular means to specify the inter-object behavior of systems (see, e.g., [12]), are included in the UML standard, and are supported by many modeling tools. In particular, we are interested in modal scenarios presented using a UML2 compliant variant of Damm and Harel's Live Sequence Charts (LSC) [9, 14], which extends the partial order semantics of sequence diagrams with universal and existential modalities and allows symbolic class level lifelines, resulting in compact and expressive specifications. The popularity and intuitive nature of sequence diagrams as a specification language in general, together with the additional unique features of LSC, motivate our choice for the target formalism of our miner. Moreover, the choice is supported by previous work on LSC (see, e.g., [16, 18, 22]), which can be practically used to visualize, analyze, manipulate, test, and verify the specifications we mine.

Our algorithm mines statistically significant LSCs of arbitrary size from program traces. Statistical significance is based on satisfaction of minimum thresholds of support and confidence (metrics adopted from data mining [11]). The algorithm leverages research in the pattern mining domain where mining is modeled as a search space exploration and efficiency is improved greatly by performing effective search space pruning strategy. The following sections describe our target specification language (i.e., live sequence charts), our mining algorithm, a case study, and a short discussion of related work. We conclude with a summary of our contributions and directions for future work.

2. MODAL SCENARIOS

We use a restricted subset of the LSC language. An LSC includes a set of instance lifelines, representing system's objects, and is divided into two parts, the *pre-chart* ('cold' fragment) and the *main-chart* ('hot' fragment), each specifying an ordered set of method calls between the objects represented by the instance lifelines. A universal LSC specifies a *universal liveness requirement*: for all runs of the

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

system, and for every point during such a run, whenever the sequence of events defined by the pre-chart occurs (in the specified order), eventually the sequence of events defined by the main-chart must occur (in the specified order). Events not explicitly mentioned in the diagram are not restricted in any way to appear or not to appear during the run (including between the events that are mentioned in the diagram). The semantics of LSC is comparable to that of various Temporal Logics [17]. For a thorough description of the language and its semantics see [9].

Syntactically, instance lifelines are drawn as vertical lines, pre-chart (main-chart) events are colored in blue (red) and drawn using a dashed (solid) line. LSCs can be visualized and edited within standard UML2 compliant modeling tools (e.g., IBM RSA [3]) using the *modal* profile [14].

3. MINING FRAMEWORK

The input for the mining algorithm are finite traces consisting of events, where each event corresponds to a triplet: caller object identifier, callee object identifier, and method signature. The output of our algorithm are statistically significant LSCs satisfying user-defined thresholds mined from the traces. We first define the statistical significance metrics considered, namely support and confidence – adopted from data mining [11]. We then describe the mining algorithm.

3.1 Witnesses, Support, and Confidence

To relate between LSCs and execution traces we first introduce notions of positive- and negative- witnesses. These are then used to compute the support and confidence values of an LSC with respect to a trace.

We consider traces to be finite words over a finite alphabet of events $\Sigma = \{a, b, c...\}$, where a unique letter corresponds to a unique triplet. We use the symbol $+\!\!+$ to represent the concatenation operator between finite words. An LSC $L(pre, main)$ defines a word m built from the concatenation of its pre-chart and main-chart finite words, i.e., $m = pre{+\!\!+}main$. For two words w, u we denote the projection of w onto the alphabet of events appearing in u by w_u. As an example, $abcadb_{ab} = abab$.

A *positive-witness* of a word w with respect to a trace T is defined as a *minimal* subword s of T such that $s_w = w$. Positive-witnesses of an LSC $L(pre, main)$ are the positive-witnesses of the word $pre{+\!\!+}main$. The set of positive-witnesses of a word w (an LSC L) with respect to a trace T is denoted by $pos(w, T)$ ($pos(L, T)$). Consider the trace $T_1 = eaeebabcedacbccdaaadabe$ as a running example. It includes two positive-witnesses of $L_1 = (ab, d)$, which are the sub-strings *abced* and *acbccd* starting at the 6th and 11th positions of the trace, respectively.

A *negative-witness* of an LSC $L(pre, main)$ with regard to a trace T, is a positive-witness of the word pre that cannot be extended to a positive-witness of L. The set of negative-witnesses of an LSC L with respect to a trace T is denoted by $neg(L, T)$. Using the previous example, T_1 includes two negative-witnesses of L_1, corresponding to the sub-strings *aeeb* and *ab* starting at the 2nd and 21st positions of the trace, respectively.

The semantics of LSC (like most formal specification languages used for reactive systems, e.g., LTL [15]) is originally defined over infinite paths. The traces we consider, however, are, of course, finite, and we do not want the arbitrary truncation of the trace to affect our confidence of the suggested universal liveness requirement specified by the LSC. We therefore need to adapt the semantics of LSC, and specifically, the definition of negative-witnesses, to finite (so called 'truncated') paths using a notion of *strong-negative-witness*. Roughly, a strong-negative-witness is negative because it explicitly violates the order specified by the main part of the LSC and not because it reaches the end of the trace. Formally, a strong-negative-witness of an LSC $L(pre, main)$ with regard to a trace T, is a positive-witness of pre, p, such that for any word w, p cannot be extended to a positive-witness of L over $T{+\!\!+}w$. The set of strong-negative-witnesses of an LSC L with respect to a trace T is denoted by $strong\_neg(L, T)$. Using the example above, note that the second negative-witness of L_1 in T_1 ends at the end of the trace and is not a strong-negative-witness.

We use the above notions of witnesses to define the statistical *support* and *confidence* metrics for LSC. Given a trace T, the *support* of an LSC $L(pre, main)$, denoted by $sup(L)$, is simply defined as the number of positive-witnesses of L found in T. The *confidence* of an LSC L, denoted by $conf(L)$, measures the likelihood of a subword in T satisfying pre to be followed by a subword satisfying $main$ or the end of the trace is reached. Hence, confidence is expressed as the ratio between the number of non-strong-negative-witnesses of the LSC and the number of positive-witnesses of the LSC's pre-chart. Formally:

$$sup(L,T) \equiv_{def} |pos(L,T)|$$
$$conf(L,T) \equiv_{def} \frac{|pos(L,T)|+(|neg(L,T)|-|strong\_neg(L,T)|)}{|pos(pre,T)|}$$

Notation-wise, when T is understood from the context, it can be omitted. Using the previous example, we have $sup(L_1, T_1) = 2$, $conf(L_1, T_1) = (2 + 2 - 1)/4 = 0.75$.

The support metric is used to limit the extraction of commonly observed interactions. The confidence metric restricts mining of such pre-chart that is followed by a particular main-chart with high likelihood. Note that LSCs with high but imperfect confidence, i.e., less than 1, are also interesting to mine (see, e.g., the notion of imperfect traces [24]), since, in general, these may reveal errors in the program or in the trace generation process (see, e.g., [6, 10, 24]). The support and confidence values for each mined LSC are included in the algorithms output.

3.2 Algorithm

We are now set to describe the basic LSC mining algorithm and sketch its soundness and completeness.

Many previous algorithms used for specification mining, e.g., [24], need to explicitly check *all possible* specifications obeying a certain template. These do not scale for specifications of an arbitrary size since the number of possible specifications is *arbitrarily large*. Rather than checking for *all possible LSCs*, we immediately *prune search spaces* containing statistically insignificant LSCs using the following property.

Property 1 (**Monotonicity of Support**). For a trace T, an LSC $L(pre, main)$, and a word w: $|pos(pre{+\!\!+}main, T)| \geq |pos(pre{+\!\!+}main{+\!\!+}w, T)|$.

Intuitively, the above property means that if a certain LSC does not meet the minimum support threshold, all its extensions will not meet the minimum support threshold.

An outline of the algorithm is given in Fig. 1. Its input includes a trace and thresholds for support and confidence, and its output is a set of LSCs. The algorithm starts by

mining a complete set of words, each having the number of positive instances greater than or equal to the support threshold. Next, it continues to compose these words into LSCs meeting the confidence threshold.

The main algorithm is given in procedure `MineLSC`, which calls the procedure `MineSupportedWords` to mine a complete set of words that meet the support threshold (line 1). `MineSupportedWords` calls `MineRecursive` to recursively add events to the current set of words in a depth first fashion. Once an extended word does not meet the support threshold (line 16), we know all its extensions will not meet the support threshold either (from Property 1), and thus we can stop recursing. After the set of words meeting the support threshold is mined, the algorithm continues to compose these words into LSCs meeting the confidence threshold (lines 3-8).

Our algorithm for LSC mining is sound and complete; i.e., not only all the output LSCs are statistically significant (i.e., meet the support and confidence thresholds), but also all the possible LSCs that are statistically significant are indeed included in the output. Soundness follows immediately from the algorithm. Completeness follows from the monotonicity property. The formal proof is outside the scope of this paper.

Procedure MineLSC
Inputs: TR : Input Trace; min_sup: Min. Sup. Thresh.; min_conf: Min. Conf. Thresh.
Output: A set of statistically significant LSCs
1: Let $WSet$ = MineSupportedWords(TR,min_sup)
2: Let $LSCResult = \{\}$
3: For every word w in $WSet$
4: For every prefix pfx of w
5: Let $main = sfx$, where $pfx +\!\!+ sfx = w$
6: Let $NewLSC$ = Create new LSC (pfx, $main$)
7: If ($conf(NewLSC) \geq min\_conf$)
8: **Output** $NewLSC$

Procedure MineSupportedWords
Inputs: TR : Input Trace; min_sup: Min. Sup. Thresh.
Output: A set of supported words
9: Let EV = Single-events appearing $\geq min\_sup$ in TR
10: Let $WSet = \{\}$
11: For every f_ev in EV
12: Call MineRecurse(TR,min_sup,EV,f_ev,$WSet$)
Return $WSet$

Procedure MineRecurse
Inputs: TR : Input Trace; min_sup: Min. Sup. Thresh.; EV: Frequent Events; $curW$: Current word considered; $WSet$: Current set of supported words
Output: Updated supported words set ($WSet$)
13: Add $curW$ to $WSet$
14: For every f_ev in EV
15: Let $nxtW = curW +\!\!+ f\_ev$
16: If ($|pos(nxtW, TR)| \geq min\_sup$)
17: Call MineRecurse(TR,min_sup,EV,$nxtW$,$WSet$)

Figure 1: Outline of the mining algorithm

The mined set of LSCs is post-processed to identify class level LSCs (see LSC *symbolic instances* [23]). In addition, we provide an array of additional user-guided filters and abstractions to further refine the resulting set of mined scenarios and reduce the complexity of the mining process. A complexity analysis of the basic algorithm and more details about its extensions are available in the technical report [21].

4. CASE STUDY

We demonstrate our approach using Jeti [4], a popular full featured open source instant messaging application. We used AspectJ to instrument the application and created trace files by recording interactions between several Jeti clients. Each of the files is approximately 1K events long (consisting of about 120 unique methods and 600 unique events). We also analyzed one long trace of 10k events. In general, the mining time for a 1K-long trace ranged between a few seconds and several minutes on a Pentium IV 3Ghz PC with 2GB memory.

Many statistically significant LSCs were mined, revealing informative scenarios. We refer interested readers to download the complete traces from [5], which also provides details on mining parameters used, runtime required, and mined results for each experiment on the traces. Two of the mined LSCs are highlighted below.

First, a mined LSC involving sending of messages when one client starts communicating with another is shown in Fig. 2 (Top). The scenario starts whenever a user uses the roster tree to select a party to communicate with. Then, the roster tree will initiate the chat and set up the chat window. After several resources and identifiers of communicating parties are obtained, eventually, an initial message is sent via the `Backend/Connect/Output` channel. Second, from traces involving the use of Jeti's group whiteboard, the miner has captured a scenario of drawing a line and sending it to the other chat users (see Fig. 2 (Bottom)).

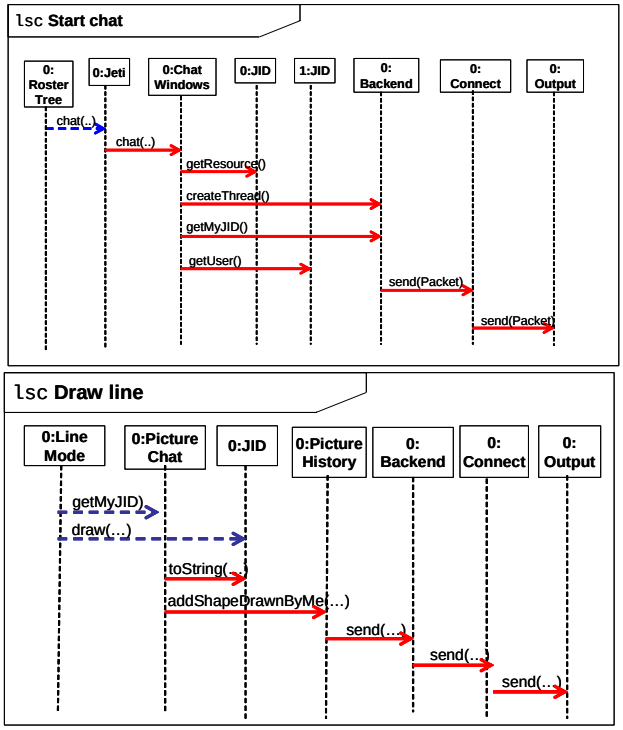

Figure 2: Mined LSCs: `Start chat` & `Draw line`

We have implemented a programmatic translation of the mined suggested LSCs (represented in simple textual format) into UML2 Sequence Diagrams, using the Eclipse UML2 APIs [2] and the *modal* profile [14]. Thus, we viewed selected results from Jeti visually inside IBM Rational Software Architect (RSA) [3] (see Fig. 3). In addition to the visual representation itself, which helped a lot in understanding the mined scenarios, we were able to use RSA to edit and manipulate the mined LSCs, group them into use cases, annotate them, print them, etc.

Finally, we used the S2A compiler [13], developed at the

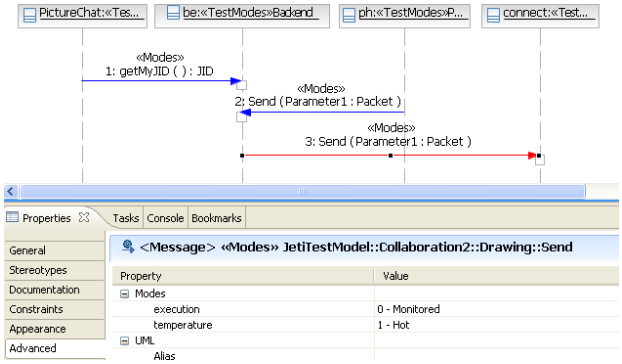

Figure 3: A mined LSC shown inside IBM RSA.

Weizmann Institute of Science, to programmatically compile selected LSCs into (monitoring) *scenario aspects* [22]. These served as scenario-based tests for Jeti and allowed us to 'validate' selected mined LSCs during subsequent executions.

Additional results and details are available in the technical report [21].

5. RELATED WORK

Most specification miners produce an automaton (e.g., [6, 8, 19]). Unlike these miners, we mine a set of LSCs from traces of program executions. Sequence diagrams in general and LSCs in particular specify inter-object behavior, where the different role of each participating object and the communications between the different objects are made explicit.

In [24], Yang et al. present an interesting work of mining two-event temporal logic rules (i.e., of the form $G(a \rightarrow XF(b))$, where G, X, and F are LTL operators [15]), which are statistically significant with regard to a user-defined 'satisfaction rate'. The algorithm presented, however, does not scale to multi-event rules of arbitrary length. To handle longer rules, Yang et al. suggest a partial solution based on concatenation of mined two-event rules. Yet, the method proposed might miss some multi-event rules or introduce superfluous rules that are not statistically significant – it is neither sound nor complete. In contrast, we mine LSCs of arbitrary size; scalability is accomplished by utilizing our own search space pruning strategy adapted from data mining domain. The method is sound and complete as all mined LSCs are statistically significant and all statistically significant LSCs are mined. The semantics of LSC includes ordering constraints which are not considered in [24].

Some work consider mining frequent patterns of software behavior (e.g., [20]). In contrast to our work, the patterns mined do not express inter-object behavior depicted using sequence diagrams.

Many work suggest and implement different variants of reverse engineering of objects' interactions from program traces and their visualization using sequence diagrams (see, e.g., [1, 7]), which may seem similar to our work. Unlike our work, however, all consider and handle only concrete, continuous, non-interleaving, and complete object-level interactions and are not using aggregations and statistical methods to look for higher level recurring scenario patterns; the reverse engineered sequences are used as a means to describe single, concrete, and relatively short (sub) traces in full (and thus may be viewed as 'existential'). In contrast, we are looking for universal (modal) sequence diagrams, which aim to abstract away from the concrete trace and reveal statistically significant recurring potentially universal scenario-based patterns, at the object-level as well as the class-level, ultimately suggesting scenario-based system requirements.

6. CONCLUSION AND FUTURE WORK

In this paper we have proposed a novel method to mine a sound and complete set of statistically significant modal scenarios from program execution traces. Our work takes advantage of the unique features of LSC as a specification language and of the available tools to visualize and use the mined specifications. The presented case study shows the utility of our approach. Our current method is limited to mining of total order LSCs. In the future, we plan to mine for additional features of sequence diagrams in general, such as explicit partial order, various structural constructs (alternatives, loops, etc.), and functional state invariants.

Acknowledgement We thank David Harel for his valuable comments and advice. We also thank Asaf Kleinbort for proofreading the final draft.

7. REFERENCES

[1] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/.
[2] Eclipse UML2. http://wiki.eclipse.org/index.php/MDT-UML2.
[3] IBM Rational Software Architect. http://www-306.ibm.com/software/rational/.
[4] Jeti. Version 0.7.6 (Oct. 2006). http://jeti.sourceforge.net/.
[5] LSC Mining: supporting material. http://www.comp.nus.edu.sg/~dlo/lscminer/.
[6] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *POPL*, 2002.
[7] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed java software. *TSE*, 32(9):642–663, 2006.
[8] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *TOSEM*, 7(3):215–249, 1998.
[9] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
[10] D.Lo and S-C.Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.
[11] J. Han and M. Kamber. *Data Mining Concepts and Techniques, 2nd Ed.* Morgan Kaufmann, 2006.
[12] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.
[13] D. Harel, A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal UML sequence diagrams. In *FASE*, 2007.
[14] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 2007.
[15] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge, 2004.
[16] J. Klose, T. Toben, B. Westphal, and H. Wittke. Check it out: On the efficient formal verification of Live Sequence Charts. In *CAV*, 2006.
[17] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *TACAS*, 2005.
[18] M. Lettrari and J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In *UML*, 2001.
[19] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.
[20] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *SIGKDD*, 2007.
[21] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenarios from program execution traces. Technical Report TRC8/07, Dept. of Computer Science, National University of Singapore, 2007.
[22] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *SIGSOFT FSE*, 2006.
[23] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *OOPSLA*, 2002.
[24] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.